НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

«До захисту допущено»

Завідувач кафедри

_____ _____
(підпис) (ініціали, прізвище)

“___” червня 20___ р.

Дипломний проєкт

на здобуття ступеня бакалавра

зі спеціальності **123 «Комп'ютерна інженерія»**

на тему:
Комплекс програм для генерації тестів за заданою граматикою

Виконав : студент (-ка) IV курсу, групи КВ-63
Бойко Владислав Володимирович
(підпис)

Керівник
к. т. н., с.н.с. Тесленко О.К.
(підпис)

Консультант з нормоконтролю, доц.каф.СПСКС, к.т.н. Клятченко Я.М. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) _____ (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Спеціальність 123 Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Романкевич В.О.
(підпис) (ініціали, прізвище)
«___»_____2020р.

ЗАВДАННЯ
на дипломний проєкт студенту

Бойку Владиславу Володимировичу
1. Тема проекту «Комплекс програм для генерації тестів за заданою граматикою»,
керівник к. т. н., с.н.с. Тесленко О.К.,
затверджені наказом по університету від 29.04.2020 р. №Х

2. Термін подання студентом проекту 20.05.2020

3. Вихідні дані до проєкту технічна література на тему теорії формальних граматик, мови програмування Common Lisp, мови програмування Java.

4. Зміст пояснювальної записки

Аналіз існуючих способів генерації тестів.

Вибір засобів реалізації.

Опис етапів розробки.

Тестування програмного комплексу.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо)

Д1. Структурна схема взаємодії модулів програмного комплексу. Схема структурна.

Д2. Структурна схема взаємодії класів об’єктної моделі парсера. Схема структурна..

Д3. Алгоритм порівняння списків об’єктів cmp-om-elem. Схема алгоритму.

Д4. Алгоритм деструктивного розкриття вкладених списків Flatten. Схема алгоритму.

6. Консультанти розділів проекту*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
нормоконтроль	к. т. н., доцент Клятченко Я.М.		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1	Видача завдання на дипломне проектування	15.12.19	
2	Аналіз існуючих рішень	27.01.20	
3	Вивчення літератури за тематикою роботи	20.02.20	
4	Вибір засобів розробки	15.03.20	
5	Розробка програмного продукту	01.04.20	
6	Підготовлення пояснювальної записки	20.04.20	
7	Оформлення матеріалів проекту	01.05.20	
8	Попередній розгляд дипломного проекту на кафедрі	20.05.20	

Студент _____ _____
(підпис) (ініціали, прізвище)

Керівник проекту _____ _____
(підпис) (ініціали, прізвище)

* Консультантом не може бути зазначено керівника дипломного проекту.

АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (51 с., 31 рис., 4 додатки).

В даній роботі досліджена проблема генерації тестів за заданою граматикою. Розглянуті і проаналізовані близькі аналоги. Приведені доводи щодо актуальності даної розробки і сфери використання, визначені і проаналізовані основні вимоги до програмного комплексу для покращення ефективності процесу розробки.

В процесі розробки були проаналізовані існуючі інструменти для парсингу на мові програмування CommonLisp, а також існуючі інструменти для створення користувацького інтерфейсу на мові програмування Java. Технічне завдання було поділене на структурні чатини для поділу розробки на етапи. Для покращення ефективності розробки була сформованя структуна схема взаємодії прогамних модулів на основі принципів архітектури MVC. Притримуючись парадигм даної архітектури модулі програмного комплексу відокремлюють реалізацію певних функціональних можливостей, що дозволяє легко додавати нові функціональні модулі.

Програмний комплекс складається з трьох основних модулів: Model, View, Controller. Модуль Model містить реалзізацію логіки програмного забезпечення, модуль View містить реалізацію користувацького інтерфейсу, модуль Controller зв’язує модулі Model і View.

Розглянуті способи формального представлення граматик. Для реалізації ядра системи за основу взята форма BNF. На вхід програмний комплекс отримує граматику вхідної мови у формі BNF. Процес генерації побудований таким чином, що на вихід подаються символьні послідовності, що повністю покривають правила вхідної граматики.

Проведено декілька етапів тестування з використання різних методів, а саме модульне тестуавання функціональних частин модуля Model,

інтеграціне тестування для перевірки взаємодії пакетів модуля Model, мануальне тествування готовго застосунку. Після проведення даних етапів, підтверджено, що програмний комплекс відповідає поставленим раніше вимогам.

Ключові слова: програмний комплекс, генерація тестів, мова програмування CommonLisp, мова програмування Java, BNF, MVC, користувацький інтерфейс, модуль, архітектура.

ABSTRACT

The qualification work includes an explanatory note (51 p., 31 pic., 4 appendices).

In this work, the topic of improving is generation of tests on the investigated grammar. Close analogues are considered and analyzed. Arguments on the relevance of this development and scope are given, the main requirements to the software package to improve the efficiency of the development process are identified and analyzed.

During the development process, the existing tools for parsing in the CommonLisp programming language were analyzed, as well as the existing tools for creating a user interface in the Java programming language. The terms of reference were divided into structural sections to divide the development into stages. To improve the efficiency of development, a block diagram of the interaction of software modules based on the principles of MVC architecture was formed. Adhering to the paradigms of this architecture, the modules of the software package separate the implementation of certain functionalities, which allows you to easily add new functional modules.

The software package consists of three main modules: Model, View, Controller. The Model module contains the implementation of software logic, the View module contains the implementation of the user interface, the Controller module connects the Model i View modules.

Ways of formal representation of grammars are considered. The BNF form is used as a basis for the implementation of the system kernel. At the entrance, the software complex receives the grammar of the input language in the form of BNF. The generation process is built in such a way that the output is a symbolic sequence that completely covers the rules of input grammar.

Several stages of testing using different methods were performed: unit testing of functional parts of the Model module, integration testing to check the interaction of Model module packages, manual testing of the finished application. After carrying out these stages, it is confirmed that the software meets the previously set requirements.

Keywords: software package, test generation, CommonLisp programming language, Java programming language, BNF, MVC, user interface, module, architecture.

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	аркушівКількість	№ прим.	Примітки
	А4	ІАЛЦ.045490.002 ТЗ	Комплекс програм для	4		
			генерації тестів за заданою			
			граматикою.			
			Технічне завдання.			
	А4	ІАЛЦ.045490.003 ТП	Комплекс програм для	2		
			генерації тестів за заданою			
			граматикою.			
			Відомість технічного			
			проєкту.			
	А4	ІАЛЦ.045490.004 ПЗ	Комплекс програм для	51		
			генерації тестів за			
			Заданою граматикою.			
			Пояснювальна записка.			
	А4	ІАЛЦ.045490.005 Д1	Структурна схема взаємодії	1		
			модулів програмного			
			комплексу.			
			Схема структурна.			

Змін.	Арк.	№ докум.	Підпи	Дата
Розробив		Бойко В.В.		
Перевірив		Тесленко О.К.		
Консульт.				
Н. контроль		Клятченко Я.М.		
Зав. каф.		Романкевич В.О.		

ІАЛЦ.045490.001 ОА

Комплекс програм для генерації тестів за заданою граматикою
Опис альбому

Літ.	Аркуш	Аркушів
	1	2

КПІ
ім. Ігоря Сікорського, ФПМ
КВ-63

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	аркушівКількість	№ прим.	Примітки
	А4	ІАЛЦ.045490.006 Д2	Структурна схема	1		
			взаємодії класів об’єктної			
			моделі парсера.			
			Схема структурна.			
	А4	ІАЛЦ.045490.007 Д3	Алгоритм порівняння	1		
			списків об’єктів			
			cmp-om-elem.			
			Схема алгоритму.			
	А4	ІАЛЦ.045490.008 Д4	Алгоритм деструктивного	1		
			розкриття вкладених			
			.списків Flatten.			
			Схема структурна.			
		Диск CD-ROM	Текст пояснювальної	1		
			записки.			
			Графічний матеріал			

Зміст

1. НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування роботи – дипломний проект на тему «Комплекс програм для генерації тестів за заданою граматикою».

Область дослідження: автоматизована генерація тестів з використанням формального представлення граматики мови.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ

Метою даної роботи є дослідження і аналіз процесу генерації тестів за заданою граматикою, які повністю покривають правила формального представлення граматики. Реалізація програмного продукту.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації в періодичних виданнях та електронні статті у мережі Інтернет.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до до програмного продукту, що розробляється:

- можливість інтеграції в існуючі проекти;
- можливість повного покриття вхідної граматики;
- можливість збереження результатів роботи;
- наявність графічного інтерфейсу користувача;

5.2. Вимоги до апаратного забезпечення:

- Оперативна пам’ять: 2 Гб;

5.3. Вимоги до програмного та апаратного забезпечення користувача:

- Операційна система, яка підтримує JVM(Java Virtual Machine).

6. ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1	Видача завдання на дипломне проектування	15.12.19	
2	Аналіз існуючих рішень	27.01.20	
3	Вивчення літератури за тематикою роботи	20.02.20	
4	Вибір засобів розробки	15.03.20	
5	Розробка програмного продукту	01.04.20	
6	Підготовлення пояснювальної записки	20.04.20	
7	Оформлення матеріалів проекту	01.05.20	
8	Попередній розгляд дипломного проекту на кафедрі	20.05.20	

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045492.004 ПЗ	Комплекс програм для	51		
			генерації тестів			
			за заданою граматикою.			
			Пояснювальна записка			
	А4	ІАЛЦ.045490.005 Д1	Структурна схема взаємодії	1		
			модулів програмного			
			комплексу.			
			Схема структурна.			
	А4	ІАЛЦ.045490.006 Д2	Структурна схема взаємодії	1		
			класів об’єктної моделі			
			парсера.			
			Схема структурна.			
	А4	ІАЛЦ.045490.007 Д3	Алгоритм порівняння	1		
			списків об’єктів			
			cmp-om-elem.			
			Схема алгоритму			
	А4	ІАЛЦ.045490.008 Д4	Алгоритм деструктивного	1		
			розкриття вкладених			
			списків.			
			Схема алгоритму.			

Змін.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045490.003 ТП		
Розробив		Бойко В.В.			Комплекс програм для генерації тестів за заданою граматикою	Літ.	Аркуш 1 / Аркушів 2
Перевірив		Тесленко О.К.					
Консульт.						КПІ	
Н.контроль		Клятченко Я.М.					

Зав. каф.	Романкевич О.В.			**проєкту**	ім. Ігоря Сікорського, ФПМ КВ-62

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
		Диск CD-ROM	Текст пояснювальної	1		
			записки.			
			Графічний матеріал.			

ЗМІСТ

ВСТУП

У наш час стрімко розвиваються комп'ютерні технологій, у сфері динамічно змінюються і створюються нові тренди. На даний момент існує велика кількість мов програмування, які застосовуються в різних сферах. Перші мови програмування беруть свій початок в 50-х роках минулого століття, з того часу помінялось багато концепцій і поглядів, але основні цілі завжди залишаються незмінними, нові технологію потребують прогресивних, швидких і якісних рішень. З розвитком сфери з'являлись нові технології, а заодно і нові мови прграмування. Оскільки була можливість реалізувати існуючі проекти за допомогою більш ефективних технологій, з'явилась така потреба. В цьому питанні провідною технологією стало використання трансляторів. У середині минулого століття транслятори активно використовувались для міграції програм між обчислювальними машинами різних моделей. Потім ці технології продовжили активно використовуватись при трансляції високорівневих мов. Транслятори залишаються актуальними і на сьогоднішній день. Базові концепції проектування трансляторів закладені в різних сферах розробки програмного забезпечення. Основні сфери застосування це звичайно ж безпосередня розробка трансляторів, програмне забезпечення для розбору текстових послідовностей і розробка мов програмування.

Для пришвидшення розробки і забезпечення вірного результату роботи вище вказаних компонентів необхідна надійна тестова система. У масштабних і сильно зв'язних програмних системах досить складно налагодити взаємодію компонентів і належне тестування, що в майбутньому призводить до безлічі проблем. На сьогоднішній день існує чимало інструментів покликаних полегшити і автоматизувати покриття програмних комплексів і їх компонентів тестами, але для покриття формальних представлень мов програмування доступного програмного забезпечення немає.

Метою даного дипломного проєкту є розробка комплексу програм, які дозволять автоматизувати тестування вище вказаних систем і забезпечити повне покриття формальних представлень цільових мов. За допомогою даного програмного комплексу можна легко генерувати тестові послідовності. Інтуїтивно зрозумілий користувацький інтерфейс значно полегшує процес знайомства розробника з доступним функціоналом. Ядро даного програмного комплексу побудоване як незалежна частина, і завдяки цьому може бути легко інтегроване з різними видами програмних комплексів, як окремий блок автоматизації тестування.

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

БНФ (Бекус-Наур форма) — формальна структура задання синтаксису, через визначення певних синтаксичних категорій з допомогою інших. Описує контекстно-вільні граматики.

ЛА — лексичний аналізатор

СА — синтаксичний аналізатор

ГК — генератор коду

ПЗ — програмне забезпечення

ГА — генетичний алгоритм

Монада — абстракція, яка описує лінійну послідовність взаємопов’язаних обчислень

IDE (Independent Development Environment) — інтегроване середовище розробки

Fazzing — метод тестування програмного забезпечення, який призначений для автоматизації виявлення помилок програмного коду

JAR (Java Archive) — звичайний ZIP-архів в якому мітиться програма на мові Java і мета-інформація для виконання програми

Junit — бібліотека для модульного тестування програмно забепечення на мові Java

TDD (Test Driven Development) — метод розробки програмного забезпечення з першочерговою розробкою тестових наборів для майбутнього функціоналу

1. ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ СПОСОБІВ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ ТЕСТІВ

Розробка програмно забезпечення — це багатофазний процес спрямований на формування певного продукту. Продукт ПЗ повинен відповідати певним нормам якості і надійності. Для забезпечення даних критеріїв програмний код повинен мати надійну систему тестування, яка гарантуватиме правильність результатів і відповідність з поставленими задачами. Оскільки на даний момент програмний код розробляється безпосередньо людьми, то не варто відкидати людський фактор. На даний момент існує чимало інструментів, які покликані спростити розробку ПЗ і його тестування. Велика кількість IDE полегшують створення типових конструкцій і допомагають розробникам максимально автоматизувати їх роботу. Фреймворки і бібліотеки для тестування максимально прискорюють процес перевірки правильності роботи ПЗ. Але оскільки процес написання тестів це не просто формування випадкової вхідної послідовності і перевірка сформованої вихідної послідовності, а й гарантувати повне покриття функціоналу, що підлягає тестуванню. Створення тестової системи з нуля — це доволі трудомісткий і довгий процес, який рідко виправдовує себе в довільних проектах, тому при процесі тестування частіше за все використовують готові інструменти для пришвидшення процесу і мінімізації витрат часового ресурсу.

1.1. Випадкова генерація

Підхід випадкової генерації тестів полягає у генерації випадкових даних, які відносяться до певних множин. Кожен згенерований тест по суті являється випадковим набором даних, а кожен набір даних вибирається з випадково підібраної множини. Тести такого виду є максимально простими в реалізації, на даний момент існує багато готових інструментів для випадкової генерації тестів. Це можуть бути як тестові фреймворки, так і онлайн сервіси. Перевагою цього підходу є швидка генерація тестових

даних великого об’єму. Але так як генерація тестових даних відбувається випадковим чином, то немає гарантії в повному покритті функціоналу, тобто деякі помилки можна пропустити , що є дуже не бажаним в процесі тестування. Звичайно існують інструменти, які дозволяють визначити, візуалізувати і дослідити покриття програмного коду тестами, але цей процес повинен виконуватись безпосередньо розробником і це впливає на головну ідею автоматизації тестування. Також значним недоліком є те, що при генерації вхідних даних, які не аналізують функціонал досить імовірною є ситуація генерації тестових послідовностей, які дублюють проходи уже покритого функціоналу, що впливає на використання часового ресурсу, котрий являється одним з ключових в сфері розробки ПЗ. Такий підхід використовують в ідеології тестування під назвою фаззинг. Фаззинг (fazzing) — це автоматизована або частково автоматизована техніка тестування програмного забезпечення, суть якої полягає в тестуванні програмного забезпечення шляхом подачі на вхід випадкових або не коректних даних. Також не раціонально використовувати даний підхід, коли предмет тестування задається великою множиною різнотипних даних.

Отже, даний підхід має вузьку сферу застосування і не дає гарантій повного покриття тестами певного функціоналу, що дискредитує його значимість в сфері автоматичної генерації тестів, винятком можуть бути конкретні реалізації даного підходу орієнтовані на специфічний функціонал. Тому в тестуванні трансляторів і подібних систем не доцільно користуватись даним підходом, хоч і вхідна послідовність даних являється рядком, але вона має певні формальні обмеження.

1.1.1. Бібліотека Randoop

Прикладом реалізації методу випадкової генерації тестових наборів є бібліотека реалізована на мові програмування Java — Randoop. Також існує версія програми, яка працює на платформі .NET. Скористуватись даною бібліотекою можна використавши JAR архів. На вхід приймається

список класів, підлягаючих під генерацію модульних тестів, а також часовий проміжок під час якого буде відбуватися генерація. Використання часового проміжку я вважаю недоцільним, тому що програма буде видавати принципово різні результати на різних машинах. Значною перевагою даного програмного комплексу являється генерація тестів, які підтримуються доволі популярним фреймворком Junit, це дозволяє працювати із згенерованими тестами простіше, а також кластеризувати процедуру запуску тестів.

Якщо розглянути приклад з офіційної документації даного програмного забезпечення, то стає очевидним, що генерація тестів даною програмою виконується досить швидко і без встановлення зайвих компонентів. Запуск генерації тестів виконується командою зображеною на рисунку 1.1.

```
java -classpath arch.jar randoop.main.Main gentests --testclass=java.util.TreeSet –output-limit=20
```

Рисунок 1.1 – Команда запуску генерації тестів

Аргументи даного запуску: arch.jar — назва архіву даного програмного забезпечення, java.util.TreeSet — клас із стандартної бібліотеки мови Java, який являється колекцією, і опція output-limit — позначає протягом якого часу будуть генеруватися тести.

Результат роботи буде збережений в класі RegressionTest.java, і тести матимуть дуже не читабельний вигляд, що значно ускладнює роботу над рефакторингом і повним покриттям цільового програмного забезпечення. Внутрішня структура тесту складається з великої кількості оголошень тестових наборів і викликів функції порівняння assert. Оскільки тести дуже об’ємні і не читабельні в прикладі наводяться тільки основні частини структури окремо взятого тесту. Анотація @Test — це стандартна анотація фреймворку Junit, завдяки якій дані тести можна легко запустити як в консолі, так і в IDE. При запуску даного програмного забезпечення на локальному комп’ютері на часовому проміжку у 10 секунд генерується в

середньому 30 тестів. Типова згенерована функція зображена на рисунку 1.2.

```
@Test
    public void test() throws Throwable {
        java.util.List<java.lang.Object> objList2 =
        java.util.Collections.nCopies(100, (java.lang.Object) (-1.0d));
        // other definitions
        org.junit.Assert.assertNotNull(objList2);
      // others asserts
}
```

Рисунок 1.2 – Згенерований Randoop тест

Отже, дана бібліотека генерації тестів справляється з завданнями тестування елементарного функціоналу, але якщо розглядати його в контексті тестування трансляторів, то його використання вважається не доцільним, тому що при розробці транслятора ми маємо жорстку прив'язку до граматики вхідної мови, а сам програмний комплекс приймає на вхід звичайну строку, тому випадкова генерація не може і не повинна гарантувати покриття тестами будь-якої частини програмного комплексу транслятора. Значним недоліком являється конфігурація роботи бібліотеки через часові параметри, тому що не можна надійно оцінити ефективність генерації тестів, оскільки часові процес є машинно-залежним. Також бібліотека не може виступати в якості частини проекту, тому що вона повністю інтегрована, що полегшує використання, але унеможливлює повний процес автоматизації конкретного ПЗ.

1.2. Генерація на основі алгоритмів пошуку

В даному підході генерація даних для тестування розглядається як задача оптимізації. Нехай існує певна функція f і якесь мінімальне значення e, яке виступає порогом функції і генерація відбувається обчисленням даної функціє, де її результат прямує до порогового значення. Очевидно, що коли порогове значення досить мале, то можна закінчити обчислення на локальному екстремумі, тим не менше дані будуть згенеровані. Також можлива генерація декількох результатів без

можливості визначення найкращого. Цей спосіб генерації можна розглядати як модифікований спосіб випадкової генерації, який не має можливості цілenапрямлено покрити логіку вкладену у певний функціонал.

Даний підхід часто використовується не тільки для генерації випадкових даних, а й для генерації модульних тестів. Додатковою перевагою даного підходу є те, що найчастіше якісною оцінкою являється показник покриття функціоналу, який виступає в якості екстремуму функції. Тому на відмінну від методу випадкової генерації тестів, де покриття функціоналу тестами можливо тільки за рахунок сторонніх інструментів, тут немає в цьому різкої необхідності, але показник строго кількісний, ніякої інформації, яка допоможе розробнику дописати не згенеровані тести не видається.

1.2.1. Бібліотека EvoSuite

Прикладом реалізації методу генерації тестів на основі алгоритмів пошуку може слугувати бібліотека написана на мові програмування Java — EvoSuite. Генерація тестів за допомогою даного програмного забезпечення відбувається на порядок довше ніж при використанні Randoop. Перевагою є те, що цільовий функціонал по можливості максимально покривається тестами. Недоліком є значний час генерації і обмеженість цільових проектів, тому що дане програмне забезпечення працює виключно з мовою програмування Java. Для перевірки роботи даного програмного забезпечення був написаний невеликий клас (рисунок 1.3) на мові програмування Java. Елементарний клас такого типу обробляється в середньому 30-40 секунд, що являється досить великим проміжком часу для обробки таких елементарних даних. Тому використання даного проекту буде одноразовим, для покриття тестами функціоналу, який існує на даний момент і при подальших змінах у ПЗ процес генерації має виконуватися по новому, що не найкращим чином впливає на ефективність розробки.

```java
public class Record {
    private String str;
    public Record(String str) {
        this.str = str;
    }
    public void printRecord(String prefix) {
        System.out.println(prefix + " " + str);
    }
    public static void main(String [] args) {}
}
```

Рисунок 1.3 – Клас для тестування роботи бібліотеки

Також програмне забезпечення підтримує тільки одну версію мови Java, а саме Java 8. Результати запуску програми видали різіні результати з різним відсотками покриття, що свідчить про нестабільність виконання. При певній кількості запусків генерації було отримано набір тестів, які повністю покривають клас Record. Вони мають структуру зображену на рисунку 1.4.

Анотація @Test — це стандартна анотація фреймворку Junit, завдяки якій дані тести можна легко запустити як в консолі, так і в IDE. Також встановлюється тайм-аут для виконання тесттів, що робить запуск більш безпечним і надійним.

```java
@Test(timeout = 4000)
  public void test0()  throws Throwable  {
      String[] stringArray0 = new String[2];
      Record.main(stringArray0);
      assertEquals(2, stringArray0.length);
  }
  @Test(timeout = 4000)
  public void test1()  throws Throwable  {
      Record record0 = new Record("XKUiLYG,j#jJ");
      record0.printRecord("(QHW&DxE5n)<A-");
  }
```

Рисунок 1.4 – Приклад згенерованого тесту

Отже, від попереднього методу генерації цей метод відрізняється не суттєво, але дає певні гарантії покриття функціоналу, які не є стабільними, але це не відміняє факту їх існування. В контексті тестування трансляторів розлгядати дану бібліотеку не доцільно, тому що тести генеруються в залежності до типів формальних параметрів класів, що підлягають тестуванню. Також суттєвим недоліком являється суттєвий час генерації і

нестабільність результатів, а також проблема з інтеграцією в проекти, що розробляються.

1.3. Генерація тестів за допомогою генетичних алгоритмів

Головна ідея генерації тестів за допомогою ГА полягає в генерації певної множини даних, тобто популяцій, далі генеруються наступні популяції за допомогою перетину уже існуючих множин, тобто відбувається операція кроссоверу, далі для кожної множини обчислюється значення цільової функції і відбувається порівняння з пороговим значенням. Даний алгоритм працює повільніше ніж класичний алгоритм пошуку, але дає можливість відійти від локального екстремуму. Значною перевагою алгоритму являється можливість отримання декількох результатів. Дана технологія дуже сильно уступає по часу виконання аналогом, тому широкого використання поки немає. В даний момент не існує програмного забезпечення в відкритому доступі для генерації модульних тестів за даною технологією.

2. ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ

Вибір засобів реалізації — це доволі не проста і важлива задача в контексті розробки ПЗ. Головне завдання зробити ПЗ максимально

універсальним і простим в користуванні. Універсальність ПЗ полягає в кросс-платфермонесті. На сьогоднішній день існує велика кількість операційних систем і ціллю даного курсового проєкту являється покрити максимальну кількість цільових машин, які можуть використовувати даний програмний комплекс. Оскільки на сьогодні одною з найпопулярніших кросс-платформенних мов програмування є мова Java, тому було вирішено використати дану мову для реалізації користувацького інтерфейсу. Для реалізації блоку парсингу доцільно використовувати функціональний підхід, а саме побудова монадних ланцюгів на основі методу парсер-комбінаторів. Існує доволі ефективна і проста у використанні система реалізована на мові програмування Lisp - SMUG. Використавши підхід реалізації Lisp систем можна з легкістю зробити ядро системи частиною інших проектів.

2.1. Способи задання граматик

Граматика — це формальний опис певних конструкцій однієї мови з використанням інших синтаксичних конструкцій. Граматика дає точну і при цьому просту для розуміння синтаксичну специфікацію мови програмування. Правильно побудована граматика мови програмування значно полегшує трансляцію вхідної програми і виявлення помилок. Граматика мови дозволяє їй ітеративно еволюціонувати, нагромаджуючи кількість нових структур і цей процес значно прискорюється, якщо мова програмування реалізована на основі свого граматичного представлення.

Для створення представлення мов програмування частіше всього використовують формальні граматики. Хоч і формальні граматики визначають тільки контекстно вільну частину мови, в контексті реалізації функціоналу даного дипломного проєкту цього буде достатньо. На сьогоднішній день одним з найпопулярніших способів задання граматик являється форма Бекуса-Наура (БНФ). При визначенні граматики за допомогою даного опису використовується два типи символів: термінальні, нетермінальні, а також металінгвістичні зв’язки. Термінальні символи —

це основні символи, які безпосередньо складають певну мову, в ролі терміналів можуть виступати цифри, літери, ключові слова, символи. Нетермінальні символи — символи, значенням якого є множина нетермінальних і термінальних символів, нетремінали пишуться в трикутних дужках “<...>”. Металінгвістичні зв’язки — це символьні розділники присоєння “::=” і логічна операція “або” (“|”) для визначення альтернатив в правилах. Загальна структура правила оголошеного за допогою БНФ зображена на рисунку 2.1, де “...” - позначає длвільну кількість елементів.

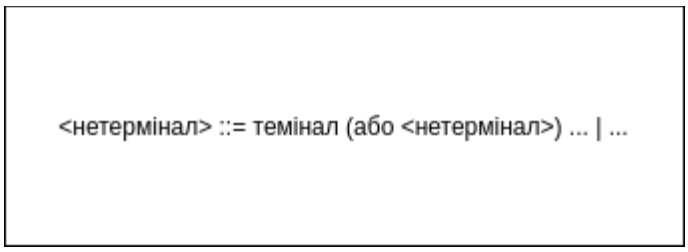

Рисунок 2.1 - Загальна структура правила БНФ

Такий спосіб досить простий у використанні, зручний для парсингу і популярний у використанні, тому в контексті реалізації ядра даного дипломного проєкту буде використовуватися саме він.

Також існує альтернативна форма запису розроблена Н. Віртом відома під назвою МБНФ (модифікована Бекус-Наур форма). Даний спосіб визначення граматики володіє ширшими можливостями ніж БНФ і також являється досить популярним, але в контексті реалізації програмного комплексу даного дипломного проєкту, було прийнято рішення використовувати БНФ, так як ціллю ПЗ являється бути максимально універсальним і доступним, а оскільки БНФ все-таки використовується в більшій кількості проектів, то це рішення виглядає цілком логічним.

Однак, пакет, який займається парсингом буде максимально відділений і незалежний від інших компонентів системи, що дозволить додати підтримку будь-якої форми задання граматик при потребі, без потреби значного внесення змін у вже існуючі модулі.

2.2. Огляд існуючих інструментів для парсингу на мові Lisp

Парсер — програма, яка з лінійної послідовності даних (лексем, символів) з урахуванням певного набору правил (граматики) будує складні структури даних. Комбінатори - це функції вищого порядку, суть яких полягає побудові однієї функції за рахунок інших.

Парсер-комбінатор — це відомий підхід створення парсерів. Суть цієї технології полягає в тому, що кожна окрема функція розглядається як окремо взятий парсер, тобто кожна функція повертає функцію парсингу. Кожен новий парсер (функція) будується на основі комбінацій інших парсерів (функцій). Парсер комбінатори частково вирішують проблеми парсингу контекстно-залежних граматик. Парсер — це не просто монада, а монада-трансформер. Де під назвою “монада” мається на увазі абстрактний ланцюг обчислень.

Парсер-комбінатори легко описуються за допомогою скінченних автоматів. Будь-який парсер має загальний вид (рисунок 2.2) типу: один вхідний стан і два вихідних стани, які позначають успішне завершення або протилежне.

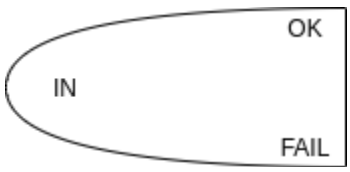

Рисунок 2.2 Загальна структура елемента парсер-комбінатора

Графи станів автоматів парсер-комбінаторів можна отримати, комбінуючи автоматів їх аргументів. Наприклад граф для парсера, який зчитує послідовність символів ‘1a’ буде мати наступний вигляд зображений на рисунку 2.3.

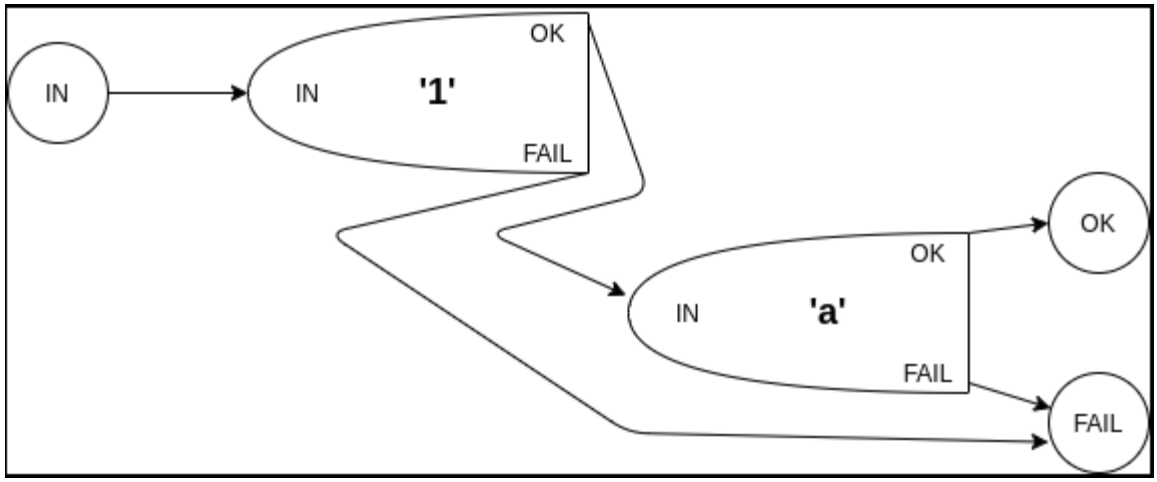

Рисунок 2.3 - Граф парсера комбінації символів “1a”

В даному дипломному проєкті буде використовуватися парсер-комбінатор написаний на основі базового парсер-комбінатора smug на мові програмування CommonLisp.

Переваги бібліотеки SMUG:

- детальна документація з прикладами використання;
- легка інтеграція з кодом написаним на Lisp;
- простота використання;
- недетермінований look ahead.

Недоліки бібліотеки SMUG:

- проблеми з відлагодженням написаних програм.

Для комбінації елементарних парсерів в більш складні необхідно забезпечити надійний спосіб комунікації між ними, в даній бібліотеці використовуються cons-пари, які мають наступний вигляд (“довільне представлення розпаршеного тексту” . “текст, що залишився”). Для максимально ефективного розбору доцільно використовувати примітивні парсери. Основні функції і їх призначення описані в таблиці 2.1.

Як видно з опису бібліотеки SMUG, вона містить основний функціонал для створення парсерів довільної складності і структури, що виглядає досить доречним в контексті реалізації програмно комплексу даного дипломного проєкту. Ряд переваг, описаних вище, зроблять

розробку максимально легкою і ефективною. Завдяки однорідній структурі всіх парсерів написаних за допомогою даної бібліотеки, досить легко виконується редагування існуючих парсерів і упрощується додавання нових. Тому в перспективі функціонал блоку парсингу даного програмного комплексу можна розширювати досить легко, що дозволить додати підтримку багатьох способів задання граматики.

Таблиця 2.1 Основні функції бібліотеки SMUG

Ім’я функції	Опис функції
.identity	Безпосередньо повертає вхідне значення у вигляді cons-пари типу (ім’я парсера . вхідне значення)
.fail	Інвертує поведінку функції .identity
.item	Повертає cons-пару типу (перший символ . Решта символів) або повертає помилку при порожньому вхідному рядку
.bind	Примітивний комбінатор для створення cons-пар з результатів виклику певних парсерів
.satisfies	Предикат, який приймає інший предикат і перевіряє його на першому символі вхідного рядка, що підлягає парсингу
.plus	Елементарний комбінатор для взаємлдії двої парсерів
.let*	Конструкція, яка забезпечує монадну структуру і проміжне зберігання результатів парсерів, а також комплексної взаємодії парсерів. Доступна можливість для ігнорування результатів роботи довільної кількості парсерів
.string=	Комбінований парсер-предикат для парсингу послідовності символів
.map	Комбінатор для забезпечення циклічного парсингу
.or	Детермінований логічний комбінатор для комбінації довільної кількості альтернативних парсерів
.not	Детермінований логічний комбінатор для визначення парсера з негативним результатом парсингу
.and	Детермінований логічний комбінатор для комбінації довільної кількості взаємопов’язаних парсерів

2.3. Огляд інструментів для створення користувацького інтерфейсу на мові Java

На сьогоднішній день існує чимало інструментів для створення користувацького інтерфейсу, кожен з яких має свої переваги і недоліки, але для реалізації блоку візуалізації даного програмного комплексу необхідно обрати ПЗ, що забезпечить максимальну сумісність з якомога більшою кількістю операційних систем, а також дозволить створити інтуїтивно зрозумілий користувацький інтерфейс.

2.3.1 Бібліотека AWT

Бібліотека AWT – це розробка компанії Sun, яка була одною з преших бібліотек для створення користувацького інтерфейсу на мові Java. Оскільки на момент створення бібліотеки мова Java не містила достатньої кількості інструментів для створення проектів такого типу, було вирішено написати основну логіку на мові прогамування C і створити обгортку на Java для розробників. Назва бібліотеки розшифровується як Abstract Window Toolkit, саме слово “абстрактний” натякає на те, що програмне забезпечення не є безпосередньою реалізацією певної логіки, а являється певним зв’язним модулем при роботі з програмним забезпеченням написаним на мові C.

Бібліотека має досить велику кількість класів і свою ієрархію (рис. 2.3.1), що є логічним, оскільки Java являється об’єктно орієнтованою мовою, тому будь-яка бібліотека, яка належить до Java API, повинна відповідати принципам об’єктно орієнтованого програмування.

Як уже відомо, що всі класи у мові Java за замовчуванням наслідуються від головного батьківського класу Object. Головним класом в ієрархіє бібліотеки AWT являється клас Component (рис. 2.4). Одна з парадигм об’єктно орієнтованого програмування: “Усе є об’єктом.”. Якщо провести аналогію з побудовою користувацького інтерфейсу, то можна сказати: “Усе є компонентом”, - якщо розглядати це в контексті бібліотеки AWT. Як видно з ієрархії класів бібліотеки, кількість компонентів не є дуже

великою, наприклад немає способу як створити таблицю, тому на даний момент цю бібліотеку майже не використовують.

Однак ряд переваг в використанні даної бібліотеки все таки є, одна з найбільших полягає в тому, що дана бібліотека є частиною JDK, це означає що розробнику достатньо просто установити мову програмування Java на свою машину (якщо це необхідно) і можна швидко створювати елементарні користувацькі інтерфейси.

Також швидкість роботи користувацького інтерфейс, створеного на основі даної бібліотеки досить висока, тому що всередині бібліотеки виконується жорсткий контроль використання ресурсів і деякі програмні модулі покликані допомогти стандартному Garbage Collector мови Java.

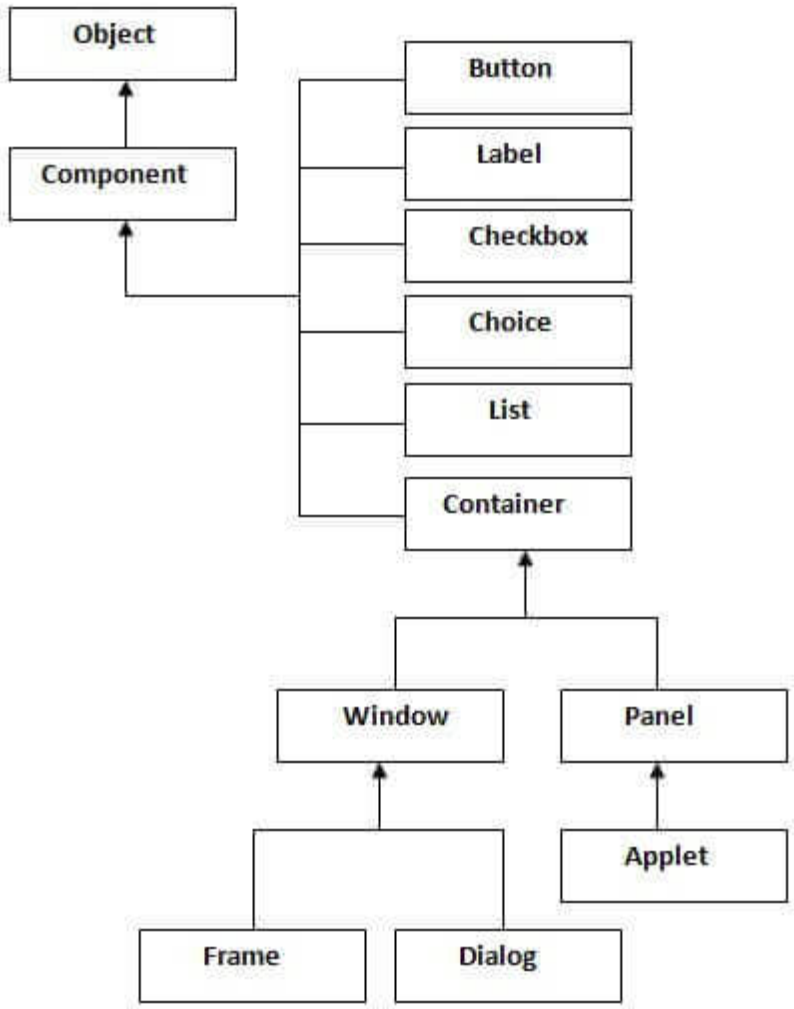

Рисунок 2.4 - Ієрархія класів бібліотеки AWT

Хоча цей інструмент моніторингу ресурсів ще не ідеальний, але компанія Oracle активно працює над оптимізацією.

Отже, проаналізувавши переваги і недоліки даної бібліотеки було вирішено не використовувати її в реалізації користувацького інтерфейсу даного дипломного проєкту.

2.3.1 Бібліотека SWING

Бібліотека Swing також являється розробкою компанії Sun, але всі компоненти і їх функціонал був реалізований виключно на мові програмування Java. Набір стандартних компонентів досить широкий і значно перевищує бібліотеку AWT. Ієрархія (рисунок 2.5) досить схожа на ієрархію класів в бібліотеці AWT, але головна відмінність полягає в тому, що компоненти бібліотеки не є системними, вони реалізовані безпосередньо всередині бібліотеки, тому розробник може легко перевизначити будь-який компонент і додати бажані властивості додатково або замінити існуючі, що значно розширює функціональні можливості проєкту.

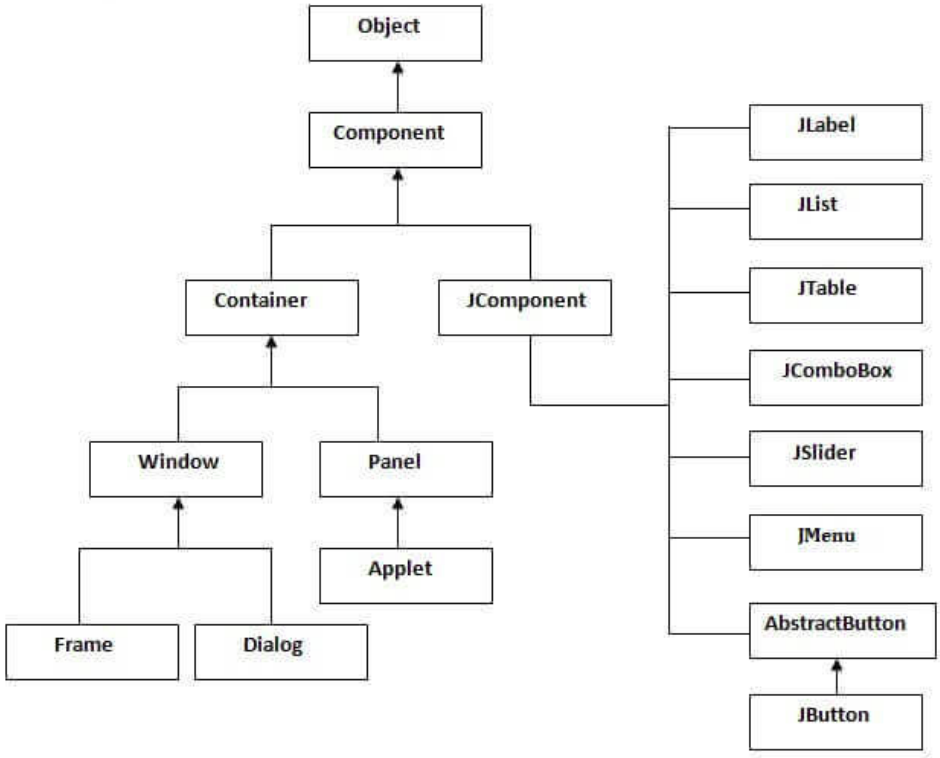

Рисунок 2.5 - Ієрархія класів бібліотеки SWING

Значною перевагою даної бібліотеки є те, що вона являється частиною JDK, тому для її використання потрібно мати на своїй машині тільки мову Java і не потрібно встановлювати додаткових компонентів. Оскільки дана технологія є дуже популярною, тому існує маса інформації по використанню і розробці користувацького інтерфейсу на її основі.

Незважаючи на всі переваги даної бібліотеки, існують також значні недоліки. Користувацький інтерфейс створений на основі даної бібліотеки не відрізняється високою швидкістю роботи, що є одним з основних критеріїв в розробці програмного забезпечення такого типу.

Отже, використання даної бібліотеки не являється найкращим варіантом для реалізації користувацького інтерфейсу даного дипломного проєкту.

2.3.3 Бібліотека JavaFx

Бібліотека JavaFx – розробка компанії Oracle, яка на даний момент вважається передовою технологією для створення користувацьких інтерфейсів. Це дуже потужна і багатофункціональна система, з масою корисних інструментів. Перший реліз даного фреймворку вийшов як частина Java 8, тому на даний момент є досить актуальною.

Ключовим поняттям в побудові інтерфейсу за допомогою даної бібліотеки є граф сцена. В ієрархіє класів (рисунок 2.6) базовим класом є Node – вузол в граф сцені, класи які його наслідують повинні задавати парметри метрики екрану. Граф сцена – це множина вузлів, які задають компоненти інтерфейсу.

Для відображення компонентів на екрані використовується графічний конвеєр, що значно прискорює роботу програми, що являється одним з головних параметрів оцінки ефективності програм графічних інтерфейсів. Зовнішній вид кожного з компонентів можна легко підкоригувати, тому що робота фреймворку інтегрована з мовою задання стилів CSS. Компанія Oracle дає доступ до документації даного фреймворку, а також велику

кількість прикладів використання. За допомогою бібліотеки можна створювати як десктопні застосунки, так і веб-застосунки.

Недоліком даної бібліотеки є те, що на даний момент ще немає остаточної версії з виправленням всіх помилок попередніх версій, але компанія Oracle активно розробляє нові релізи, тому це не являється вагомим недоліком.

Отже, дана бібліотека доволі потужний і зручний для розробки користувацького інтерфейсу інструмент, тому в конексті вирішення проблеми побудови користувацького інтерфейсу для даного дипломного проєкту буде використовуватися саме він, тому що він задовільняє всі початкові умови. Інструментарій даної бібліотеки дозволить побудувати доволі простий в користуванні і водночас швидкий користувацьки інтерфейс.

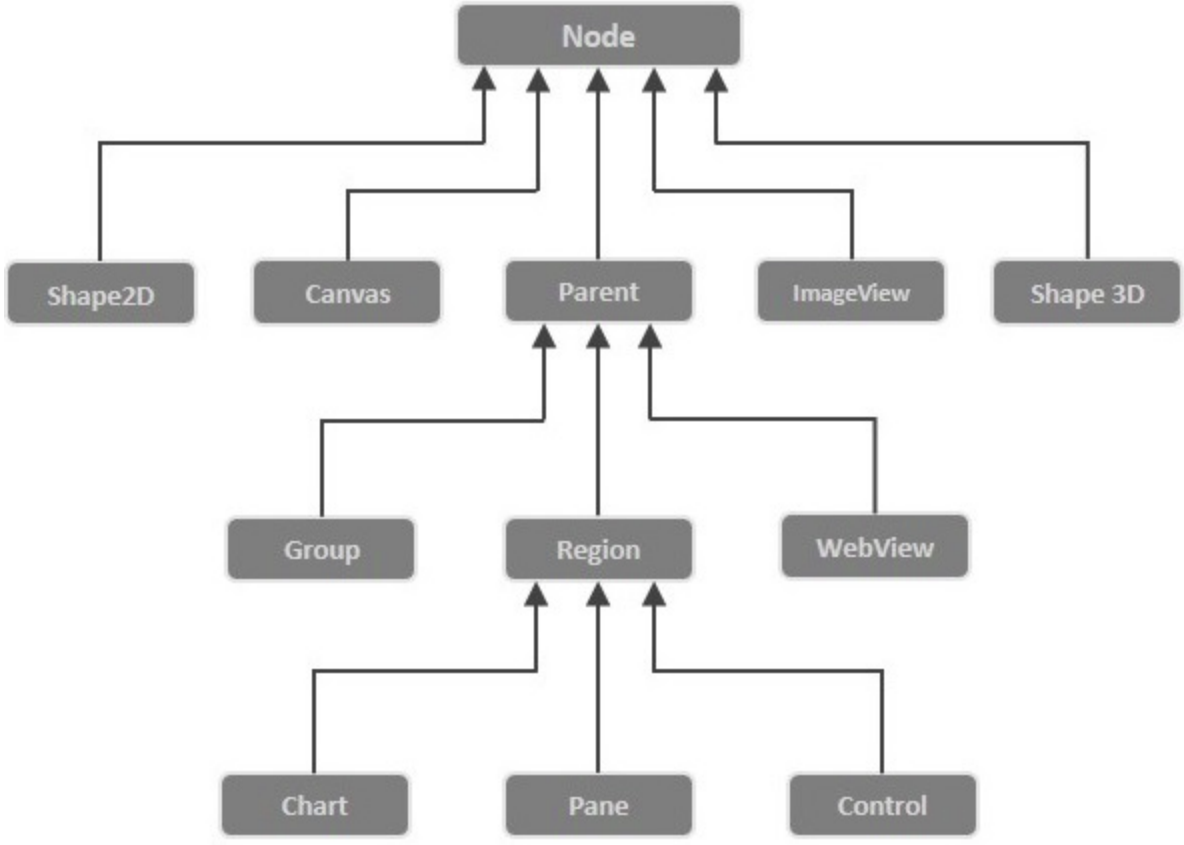

Рисунок 2.6 - Ієаррхія класів бібліотеки JavaFx

3. РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ

Розробка програмних засобів — це складний, багатокомпонентний процес, який складається з певної послідовності дій, результатом яких є готовий продукт.

Основні компоненти процесу розробки ПЗ:

- специфікація вимог;
- проєктування;
- реалізація;
- тестування;
- впровадження;
- супровід.

Всі компоненти є невід'ємною частиною розробки ПЗ і чітке слідування етапам допоможуть структуризувати роботу і спрогнозувати терміни виконання.

3.1. Розробка структурної схеми програмного комплексу

Розробка структурної схеми є важливим компонентом етапу проєктування. При побудові структурної схеми одночасно розробляється план майбутніх робіт і розподіл задач. Оскільки в контексті розробки програмного комплексу даного дипломного проєкту всі етапи виконуються одним розробником, тому стадію розподілу задач можна упустити.

Програмний комплекс буде складатися із трьох основних модулів: модуль користувацького інтерфейсу, контролер, модуль реалізації логіки.

Така структура описується паттерном проектування MVC (Model View Controller), це досить популярний підхід для створення як ПК-застосунків, так і веб-застосунків. Суть підходу полягає у максимальній структуризації програмного коду. Користувач безпосередньо взаємодіє з інтерфейсом за який відповідає блок View, після виконання певної дії користувачем посилається відповідний сигнал на Controller, який в свою чергу вирішує якому з блоків, який відповідає за певну задачу потрібно доручити дану задачу.

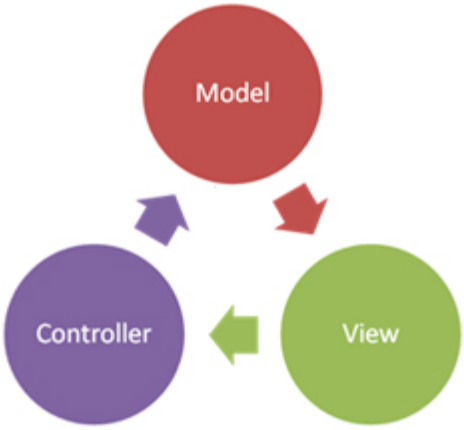

Рисунок 3.1 Структурна схема паттерна проектування MVC

Компоненти, що реалізують певну логіку, розміщуються в блоці Model. Зазвичай реалізація кожного з блоків доручається різним розробникам, або й різним командам командам розробників.

Блок View включає в себе незалежні елементи побудови користувацького інтерфейсу, які зображені на рисунок 3.2.

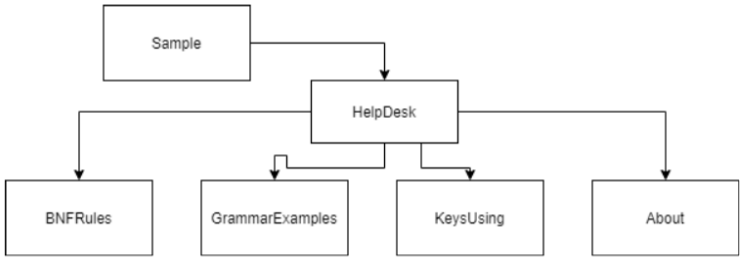

Рисунок 3.2 - Структура блоку View

Блок реалізації має доволі просту і зрозумілу структуру, всі додаткові елементи повинні або задаватися в блоці sample, або бути його підкомпонентами.

Як було уже зазначено вище, для реалізації блоку користувацького інтерфейсу використовується бібліотека JavaFx. Дана бібліотека дозволяє будувати користувацький інтерфейс трьома різними способами:

- За допомогою програмного коду на мові Java
- За допомого інструменту SceneBuilder
- За допомогою конфігураційних файлів формату .fxml

Проаналізувавши всі варіанти було вирішено використовувати третій підхід, це дозволить швидко і ефективно виконати задачу побудови користувацького інтерфейсу, а також забезпечить розділення програмних блоків з створенням мінімальних залежностей.

У якості контролера використовується клас реалізований на мові Java, під назвою Controller. В реалізації даного класу немає ніяких структурно складних елементів. По суті цей клас це просто набір методів, які тригеряться під час виконанням користувачем певних дій. Блок генерації тестів використовується у вигляді бінарних файлів, тому контролер просто формує запит від користувача і відсиляє його в блок генерації тестів, і виконує певні дії для відображення результуючих даних.

Структурна схема блоку Model має більшу кількість компонентів, а відповідно і більш складну структуру.

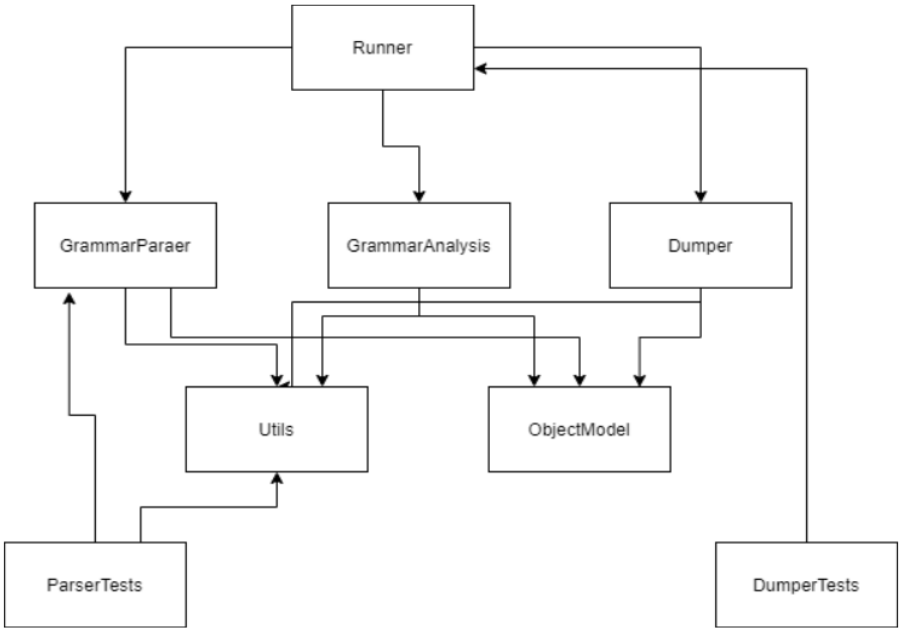

Рисунок 3.3 - Структура блоку Model

Для максимально простої взаємодії між блоками було прийнято рішення об’єднати всі трансформації в блоці Runner і створити єдину інтерфейсну функцію для взаємодії з блоком Controler.

Повна схема взаємодії всіх блоків буде представлена в графічній частині роботи.

Отже, завдяки побудованій схемі можна зробити процес розробки значно ефективнішим. Структурна схема дозволить розбити основну задачу на більш малі підзадачі і розподілити розробку на існуючому часовому проміжку.

3.2. Опис блоків реалізації логіки програмного комплексу

Блок, який відповідає за реалізацію логіки програмного комплексу написаний на мові програмування Lisp. Всі залежності між компонентами задаються у файлі (рисунок 3.2) системи tests-generator.asd. Система збору програмних пакетів ASDF (Another System Definition Facility) відіграє роль своєрідної точки запуску, що допомагає запустити роботу блоку однією командою. Команда (asdf:load-system :tests-generator) запускає компіляцію

всієї системи і дозволяє використовувати окремі функції пакетів, які знаходяться в даній системі. Повний процес генерації тестів можна розділити на декілька основних компонентів: виконання парсингу вхідної граматики, проведення аналізів над внутрішнім представленням граматики, генерації символьних послідовностей із внутрішнього представлення граматики.

Як видно на рисунок 3.4, існує три незалежних компоненти: smug — це пакет, який містить основний функціонал для побудови монадних парсер-комбінаторів, object-model — містить класи, які після парсингу відіграватимуть роль внутрішнього представлення вхідної граматики, utils — пакет з типовими функціями різнотипних обходів довільних структур даних.

```
(asdf:defsystem :tests-generator
  :description "System that unite packages for tests generation."
  :author "Vladyslav Boiko"
  :components ((:file "smug")
               (:file "object-model")
               (:file "utils")
               (:file "parser" :depends-on ("smug" "object-model" "utils"))
               (:file "parser-tests" :depends-on ("parser" "object-model"))
               (:file "grammar-analysis" :depends-on ("object-model" "utils"))
               (:file "dumper" :depends-on ("object-model" "utils"))
               (:file "runner" :depends-on ("object-model" "parser" "grammar-analysis" "dumper"))
               (:file "dumper-tests" :depends-on ("runner"))))
```

Рисунок 3.4 Система взаємодії компонентів на мові Lisp

3.2.1 Блок парсингу

Блок парсингу знаходиться в пакеті :parser. Єдина функція, яка доступна із цього пакету називається parse-input і приймає на вхід строку вхідної граматики і пусту хеш-таблицю для збереження проміжних результатів парсингу. Пакет залежить від об’єктної моделі (рисунок 3.3), яка виступає в ролі проміжного представлення правил вхідної граматики.

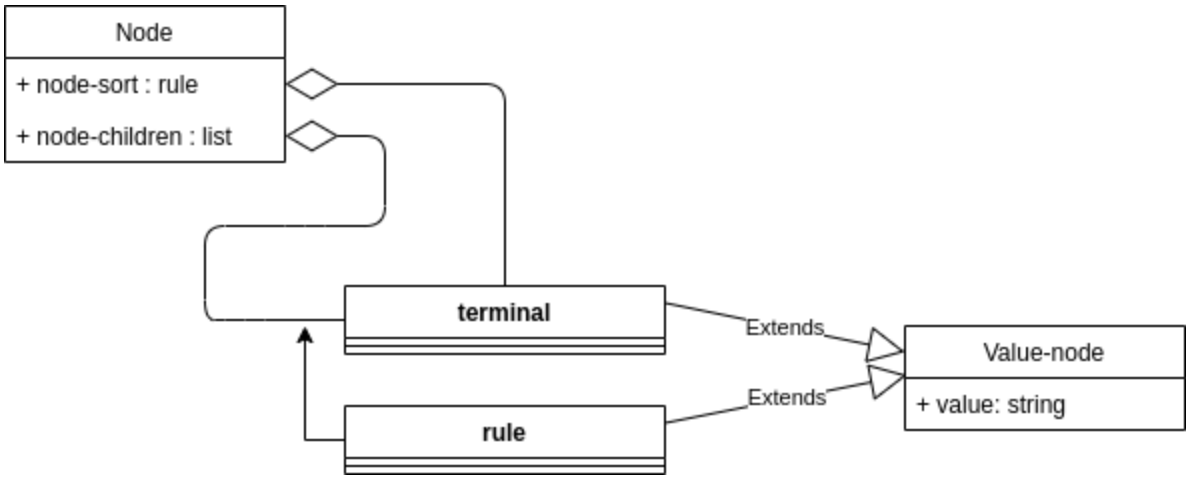

Рисунок 3.3 - Ієрахія класів об’єктної моделі

Структуру правила БНФ було описано в попереднтому розділі. Після парсингу кожне правило перетвориться в об’єкт Node. Ім’я павила буде записано в полі node-sort, а вміст буде знаходитися в полі node-children, що являє собою спискову структура, елементами якошо будуть об’єкти terminal і/або rule, які в свою чергу містять просто строкове представлення розпаршеної символьної послідовності. Загальна структура парсер комбінатора граматики представлена в таблиці 3.1.

В таблиці 3.1 описані основні функції, які демонструють перетворення вхідної граматики в певну об’єктну модель.

Таблиця 3.1. Опис структури парсер-комбінатора вхідної граматики

Назва парсера	Компоненти	Опис
.syntax	.bnf-rule	Початковий парсер який комбінує роботу одного або кількох парсерів .bnf-rule за допомогою парсер комдінатора .one-or-more
.bnf-rule	.rule, .assigment,	Основна функція парсингу правила, яка створює об’єкт Node і зберігає

	.expression	допоміжну інформацію в таблицю правил
.rule	.rule-name	Виконує парсинг нетермінального символу “<...>” і створює об’єкт rule
.assigment	.string=	Виконує парсинг опрації “::=” і повертає істину без збереження результату
.expression	.or-list	Комбінує роботу довільної кількості парсерів .or-list, але не менше одного
.or-list	.list	Виконує комбінацію довільної кількості парсерів .list, які розділені між собою символом “|”
.list	.term	Виконує комбінацію одного або декількох парскрів .term і повертає результат їх роботи у вигляді списку
.term	.rule, .literal	Виконує комбінацію двох парсерів .rule і .literal за допомогою логічного предикат-парсера .or
.literal	.text	Виконує виклик парсера .text, який повертає набір символів, що знаходиться в лапках і повертає об’єкт rule, який містить дане значення

Якщо розглянути загальну структуру функції парсер-комбінатора, то видно, що процес парсингу відділяється від повернення результату, що дозволяє використовувати інші види внутрішніх представлень вхідної граматики, без масштабних змін.

Структура функції парсера досить проста (рисунок 3.4), використовується комбінатор .let*, в якому позначаються результати саб-парсерів, які потрібно зберегти, і які потрібно ігнорувати. Елементи, які ігноруються, позначаються символом “_”.

```
(defun .rule ()
  (.let* ((_ (.whitespace?))
          (_ (.char= #\<))
          (rule-name (.rule-name))
          (_ (.char= #\>)))
    (.identity (make-instance 'rule
                              :value rule-name))))
```

Рисунок 3.4 - Загальна структура типової функції парсер комбінатора

Основна функція блоку має назву parse-input (рисунок 3.5), вона виконує операцію замикання вхідної таблиці, яка оголошена в якості глобального символу, це дозволяє вільно користуватися даною таблице всередині пакету. Функція повертає два значення: список оброблених правил у вигляді об’єктів, а також допоміжну таблицю правил. Допоміжна таблиця має наступну структуру: ключ – ім’я правила, значення – вираз у вигляді об’єктної моделі.

```
(defun parse-input (input rules-table)
  (let ((*rule-tables* rules-table))
    (values (parse (.syntax) input) *rule-tables*)))
```

Рисунок 3.5 - Інтерфейсна функція пакета парсингу

Оскільки хеш-таблиця не забезпечує точний порядок слідування елементів, то основний обхід буде відбуватися по результуючому списку, але обробка рекурсивних правил використовуватиме допоміжну таблицю правил, оскільки пошук елементів в хеш таблиці відбувається за O(1), на вдімінну від O(N) при використанні лінійного пошуку в списку. На дуже малих вхідних граматика час виконання не буде відрізнятися, і різниця у використанні пам’яті не буде суттєвою, але на великих проміжках час виконання при використанні пошуку у хещ таблиці дасть значний виграш у швидкодії, але й невеликий програш у використанні пам’яті. Але, оскільки, зазвичай граматикаи мов програмування складаються з приблизно 100-500 правил, то це дає значний виграш у швидкодії і незначний програш по пам’яті.

Отже, проаналізувавши реалізацію пакету парсингу можна зробити висновок, що завдяки застосуванню методології парсер-комбінаторів програмний код є доволі простим і структурованим, а використання

ресурсів являється доцільним і обгрунтованим, що в перспективі позитивно вплине на швидкодію програмного забезпечення, що використовує даний код.

3.2.2. Блок аналізу об’єктної моделі граматики

Для виконання генерації тестів за заданою граматикою, виконується парсинг вхідної граматики, після чого отримано об’єктне представлення даної граматики, але для легкої і швидкої генерації тестових наборів потрібно привести об’єктне представлення в один об’єкт, який містить всі можливі способи обходу граматики і при цьому повністю зберігає правильні позиції елементів. Нормалізація об’єктного представлення граматики знаходиться в пакеті grammar-analysis. Ім’я інтерфейсної функції run-rules-transforamtion.

Дана функція вирішує проблему нормалізації рекурсивних правил. Нехай існує правило, яке зображене на рисунок 3.6, тоді існує два набори символі, які покривають дане правило: “term1 , term2”, “(term1 , term2)”. Навіть якщо правило містить інші альтернативи, для розкриття даного правила буде достатньо підставити хоча б одну нерекурсивну альтернативу даного правила, тому що головною задачею програмного комплексу даного дипломного проєкту є максимальне покриття вхідної граматики.

Після нормалізації дане правило (рисунок 3.7) матиме вигляд придатний до наступної фази процесу генерації тестів.

```
<rule> ::= 'term1' ',' 'term2' | '(' <rule> ')'
```

Рисунок 3.6 - Рекурсивне правило записне у БНФ

```
<rule> ::= 'term1' ',' 'term2' | '(' 'term1' ',' 'term2' ')'
```

Рисуное 3.7 - Не рекурсивне правило записане у БНФ

Функція run-rules-transformation зображена на рисунок 3.8. Дана функція не є звичайною лінійною функцією, в її тілі замкнуті декілька функцій, які виконують проміжні трансформації.

На вхід дана функція отримує допоміжну таблицю, яка була сформована в процесі парсингу. Використання хеш-таблиці прискорить доступ до елементів при їх редагуванні. Основним тілом функції являється цикл, який проходить по всім правилам і перевіряє чи дійсно вони рекурсивні, ця перевірка виконується за допомогою вкладеної функцій %check-recursive-appearance, яка на вхід приймає пару ключ/значення поточного елемента таблиці, а також нерекурсивну альтернативу для підстановки. Пошук нерекурсивної альтернативи для підстановки виконується вкладеною функцією %find-non-recursive, також дана функція зупиняє аналіз і повертає помилку, якщо не було знайдено жодної нерекурсивної альтернативи.

```
(defun run-rules-transformation (rules-table)
  (labels ((%check-recursive-appearance (node-list rule-name non-rec-rule)
             (mapcar (lambda (rl-lst)
                       (flatten
                        (mapcar (lambda (rl-item)
                                  (if (and (typep rl-item 'rule)
                                           (string= (value rl-item)
                                                    rule-name))
                                      non-rec-rule
                                      rl-item))
                                rl-lst)))
                     node-list))
           (%find-non-recursive (node-list rule-name)
             (let ((res (remove nil
                                (mapcar (lambda (rl-lst)
                                          (unless (find-if (lambda (rl-item)
                                                             (and (typep rl-item 'rule)
                                                                  (string= (value rl-item)
                                                                           rule-name)))
                                                           rl-lst)
                                            rl-lst))
                                        node-list))))
               (if res
                   (first res)
                   (error "Rule is very recursive.")))))
    (loop :for key :being :the hash-keys :of rules-table
          :using (hash-value value)
          :do (let ((non-rec-rule (%find-non-recursive value key)))
                (setf (gethash key rules-table)
                      (%check-recursive-appearance value key non-rec-rule))))
```

Рисунок 3.8 - Функція run-rules-transformation

Отже, можна зробити висновок, що даний блок виконує проміжні аналізи над об’єктним представленням вхідної граматики для спрощення процесу генерації тестів в пакеті :parser, який приймає на вхід результат роботи даного функціонального блоку.

3.2.3. Блок генерації тестових послідовностей

Формування тестових послідовностей виконується в пакеті dumper, який відповідає за приведення об’єктної моделі до виду придатного для генерації символьних послідовностей і безпосередньо за генерацію списку таких послідовностей. Інтерфейсна функція get-result-list отримує на вхід ім’я аксіоми граматики і допоміжну таблицю, яка була сформована в блоці парсингу і відредагована в блоці аналізу об’єктної моделі граматики.

Таблиця замикається в глобальний символ, для зручного доступу в межах даного пакету. Тіло функції — це монада (рисунок 3.9), і має вона наступний вигляд: (form-result-list (flatten-terminals-list (table-traverse first-rule))). Після ланцюга взаємопов’язаних трансформацій ми отримує результуючий список.

Функція table-traverse виконує згортання всіх нетерміналів, які знаходяться у виразі аксіоми, тобто якщо в нас є правило виду:

<rule1> ::= <rule2> (1),

то на місце <rule2> підставляється вираз правила <rule2>. Нехай існує правило:

<rule2> ::= ‘1’ (2),

то після виконання функції table-traverse над об’єктним представленням правила (1) над його виразом буде виконана операція інлайнингу і воно матиме наступний вигляд: <rule1> ::= “1”.

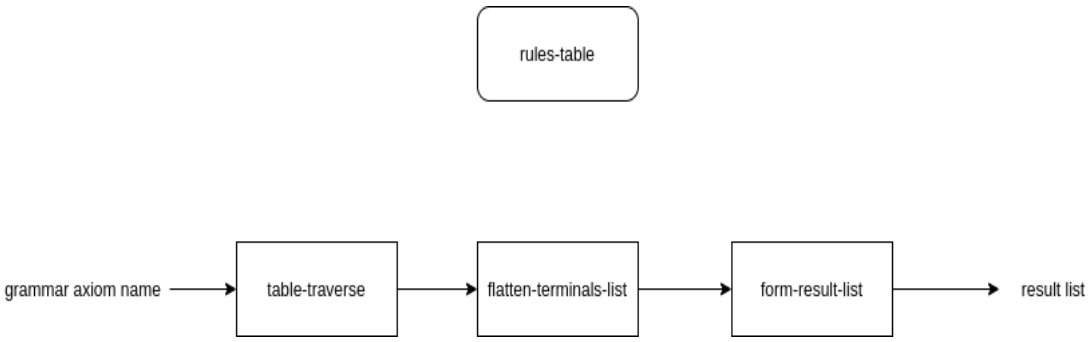

Рисунок 3.9 - Структура монади формування результуючого списку

Обхід об’єктного представлення виразу виконується як обхід списку з рекурсивним викликом, якщо поточний елемент, що оброблюється, є об’єктом класу rule. Результатом роботи даної трансформації є списком нетерміналів з урахуванням обробки альтернатив виразів. Якщо виконати дану трансформацію для правила, яке зображене на рис. 3.6, то в результаті буде отримано список, який складається із двох списків об’єктів класу terminal. Перевагою використання даного підходу є висока швидкодія, тому що для інлайнингу об’єкта класу rule виконується пошук його виразу, а оскільки всі виразу розміщені в допоміжній таблиці правил, то доступ до елементів виконується за O(1). Дана функція може бути універсальною,

для легшого розширення системи, якщо в якості аргументу передати тип класу в об’єктах якого знаходяться значення нетреміналів.

Оскільки об’єктні представлення альтернативних виразів кожного правила зберігаються у вигляді спискових структур, то доцільно було б розкрити такі списки, щоб результуючі списки мали лінійну структуру. Цю операцію виконує функція flatten-terminals-list.

І наприкінці ланцюга операцій монади виконується функція form-result-list, яка безпосередньо обходить список елементами якого є списки об’єктів класу terminal і зберігають їх значення у вигляді звичайних рядків. Роздільник за замовчуванням символ пробілу.

Отже, монадна структура даного пакету дозволяє легко додавати проміжні аналізи і трансформації для розширення функціоналу програмного комплексу.

3.3. Опис контролера

Контролер є основним елементом моделі MVC, призначенням якого є зв’язування користувацького інетерфейсу з блоком реалізації логіки програми. В даному програмному комплексі Controler представлений у вигляді одного класу, який містить набір функцій що реагують на певні дії коритстувача.

Бібліотека JavaFx дозволяє виконувати зв’язування елементів користувацького інтерфейсу і функцій конторлера двома способами:

- За допомогою анотації @FXML
- За допомогою атрибуту onAction, графічних компонентів

В даному проєкті було вирішено комбінувати дані підходи, і для клавіш використовувати атрибут onAction, а для текстових полів анотацію @FXML. Завдяки такому розділенню можна легко зрозуміти, які компоненти залежать від результату роботи логіки програми, а які можуть міняти свій стан тільки в контролері.

Перевагою використання такого підходу є можливість внесення змін в кожен з блоків незалежно від інших, звичайно, якщо це не концептуальні зміни.

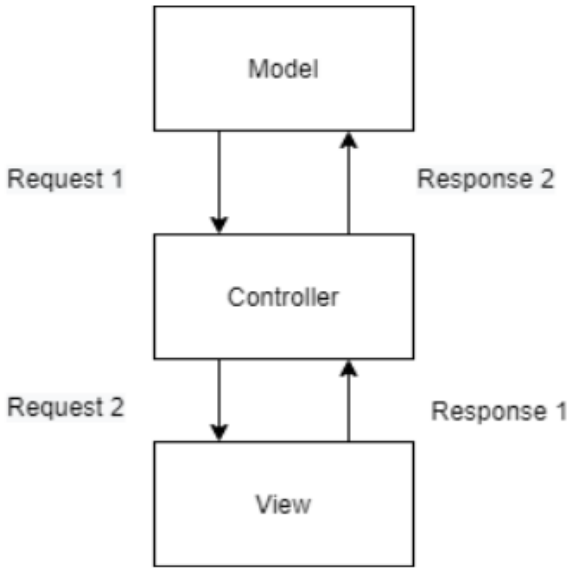

Рисунок 3.10 - Взаємодія блоків системи з контролером

Взаємодія інших блоків з контролером зображені на рисунку 3.10, і процес взаємодії відбуваються наступним чином:

1. Користувач здійснює певну дію і відправляється запит Request 1 на контролер
2. Контролер виконує обробку вхідного запиту і формує запит Request 2 і відправляє його на блок Model
3. Блок Model виконує обробку вхідного запиту, виконує певні дії, формує відповідь Request 1 і відправляє її на контролер
4. Контролер отримує відповідь, обробляє її, якщо це потрібно і виконує певні дії для відображення результату у користувацькому інтерфейсі, тобто формує своєрідну відповідь Response 2.

3.4. Опис блоків користувацького інтерфейсу

Користувацький інтерфейс реалізований з використанням бібліотеки JavaFx. Всі конфігурації і елементи створюються у файлах fxml (рисунок 3.11). Макет екрану GridPane. Оскільки будується двовимірний інтерфейс то було вирішено використовувати макет GridPane. Даний макет представляє вікно програми у вигляді своєрідної матриці. Кожен елемент займає певну комірку, або декілька комірок даної матриці. Такий підхід дозволяє швидко побудувати користувацький інтерфейс, без зайфих конфігурацій форматування екрану.

Для того, щоб отримати доступ до певного елементу користувацького інтерфейсу в інших модулях системи, йому потрібно присвоїти певний id, це виконується командою fx:id=”…” (рисунок 3.11).

```
<Button mnemonicParsing="false" onAction="#helpDesc" text="?" textAlignment="CENTER" GridPane.columnIndex="0" GridPane.rowIndex="0" />
<Button mnemonicParsing="false" onAction="#loadFile" text="Load" textAlignment="CENTER" GridPane.columnIndex="0" GridPane.rowIndex="3" />
<Button fx:id="generateBtn" mnemonicParsing="false" text="Generate Tests" GridPane.columnIndex="0" GridPane.rowIndex="4" />
<Button fx:id="saveIntoFileBtn" mnemonicParsing="false" onAction="#saveIntoFile" text="Save" GridPane.columnIndex="0" GridPane.rowIndex="5" />

<TextArea fx:id="fileLoadArea" prefHeight="300.0" prefWidth="400.0" GridPane.columnIndex="1" GridPane.rowIndex="1" />
<TextArea fx:id="resultArea" prefHeight="300.0" prefWidth="400.0" GridPane.columnIndex="1" GridPane.rowIndex="2" />
```

Рисунок 3.11 - Приклад створення елементів користувацького інтерфейсу

Доступ до елементів в інших модулях системи виконується за допомогою використання анотацій бібліотеки JavaFx (рисунок 3.12).

```
@FXML
private TextArea fileLoadArea;

@FXML
private TextArea resultArea;
```

Рисунок 3.12 - Доступ до елементів користувацького інтерфейсу в довільному модулі системи

Детальний опис структури користувацького інтерфейсу знаходиться в розділі 3.1.

В таблиці 3.1 описані функціональні призначення компонентів блоку View.

Таблиця 3.1 Функціональний опис компонентів блоку View

Назва блоку	**Функціональне призначення**
Sample	Реалізації головно вікна користувача
HelpDesk	Реалізація блоку підказок для користувача
BNFRules	Блок підказок по коректному оформленню вхідної граматики
GrammarExamples	Блок, який відображає користувачу основні приклади правил вхідної граматики, які підтримуються даним ПЗ
KeyUsing	Опис роботи клавіш головного вікна
About	Короткі відомості про програмний продукт і розробника

Отже, блоки користувацького інтерфейсу мають досить просту структуру, і невелику кількість елементів, що дозволяє зробити користувацький інтерфейс максимально лаконічним і зрозумілим для користувача. Даний модуль інтегрований від інших модулів, тому внесення змін і розширення функціональних можливостей програмного комплексу буде виконуватися швидко і ефективно.

4. ТЕСТУВАННЯ ПРОГРАМНОГО КОМПЛЕКСУ

Тестування програмного комплексу важлива стадія процесу розробки програмного комплексу. За допомогою належної тестової системи можна чітко оцінити відповідність результату роботи ПЗ до поставленої задачі. Оскільки варіантів написання тестів дуже багато, а кількість можливих тестів навіть для невеликого програмного комплексу дуже велика, потрібно раціонально розділити методи тестування і блоки, які підлягають тестуванню для максимально ефективного тестування опираючись на реальну кількість людського і часового ресурсів. Існує багато видів і підходів до виконання тестування ПЗ, основні з них зображені на рис 4.1.

Рисунок 4.1 - Ієрархія методів тестування

У контексті виконання тестування даного програмного комплексу було вирішено провести три основних етапи тестування, кожен з яких може ділитися на менші частини:

1. Модульне тестування блоку парсингу
2. Інтеграційне тестування backend-частини програмного комплексу
3. Манульне тестування взаємодії всіх компонентів системи

Даний підхід дозволить максимально ефективно оцінити роботопридатність програмного комплексу з урахуванням існуючого часового і людського ресурсів. Варто зауважити, що новий етап тестування не може бути розпочатий допоки попередній не закінчений, тому що це може привести до марної втрати існуючих ресурсів, що в загальному вплине на результат роботи в негативну сторону.

4.1. Опис та результати модульного тестування

Модульний тест (unit test) — програмний виклик певного функціоналу, що підлягає тестуванню, з метою гарантування коректності результату роботи шляхом порівняння з очікуваним результатом. Особливістю модульного тестування є те, що перевіряється робота конкретного модулю системи, це може бути окремо взята функція або певний набір функцій. Модульні тести допомагають розробникам швидко

перевіряти коректність роботи певного функціоналу, а також швидко вносити зміни у готові проекти з мінімальними ризиками, так звана регресивна розробка. Модуль в контексті тестування можна описати як невелику, зв'язну частину коду, функціональним призначенням якого є реалізація конкретної поведінки.

Модульні тести можуть виступати в якості документації до коду, тому що кожен тест є описом конкретної частини функціоналу конкретного програмного комплексу. Останнім часом набуває популярності підхід розробки програмного забезпеченням шляхом попереднього написання тестів, ця концепція називається TDD (test driven development). Це дозволяє добре розібратися зі структурою і вимогами майбутнього програмного забезпечення.

В ієрархії підходів тестування (рисунок 4.1) модульне тестування є фундаментальним, тобто нехтувати ним, в проектах будь-якого масштабу вважається не доцільним.

Для реалізації модульного тестування на мові Lisp було реалізовано тестову систему на основі макросів. Цей підхід забезпечив швидку і компактну реалізацію тестових блоків. Дана тестова система дозволяє запускати всі тести і отримувати статистику тестування певного блоку в компактному вигляді. Також є можливість запускати окремо взятий тест для зручного відлагодження функціоналу що підлягає тестуванню.

Модульні тести для блоку парсингу розміщені в пакеті :parsr-tests. Як видно на рисунку 4.2, типовий тест є досить компактним і простим для розуміння.

Вхідними параметрами макросу def-test є: ім'я тестової функції, рядок з граматикою і очікуване об'єктне представлення вхідної граматики. Всі тестові функції обчислюються на стадії компіляції і зберігаються в хеш-таблиці у вигляді: ключ - ім'я функції, значення lamda-функція виклику блоку парсингу.

Результат запуску всіх тестів зображений на рисунку 4.4., як видно статистика видається в компактному вигляді, і якщо в списку тестів не всі виконаються успішно, то їх імена будуть показані в рядку “Failed tests list:” і них можна буде запустити окремо за допомогою функції run-test.

```
(def-test test.1
  "<A> ::= <B> | <C>
<B> ::= '1'
<C> ::= '2'
"
  (list (make-instance 'node
                       :node-sort (make-instance 'rule
                                                 :value "A")
                       :node-children (list
                                       (list (make-instance 'rule
                                                            :value "B"))
                                       (list (make-instance 'rule
                                                            :value "C"))))
        (make-instance 'node
                       :node-sort (make-instance 'rule
                                                 :value "B")
                       :node-children (list
                                       (list (make-instance 'terminal
                                                            :value "1"))))
        (make-instance 'node
                       :node-sort (make-instance 'rule
                                                 :value "C")
                       :node-children (list
                                       (list (make-instance 'terminal
                                                            :value "2"))))))
```

Рисунок 4.2 - Типовий модульний тест із модуля parser-tests

```
CL-USER> (parser-tests:run-all-tests)
All parser tests runner:
+++++++

All tests was executed:
Passed: 7;
Failed: 0;
Failed tests list:
T
```

Рис. 4.3 Результат запуску всіх тестів модуля parsee-tests

Функція run-test видасть більш детальну інформацію про запуск тесту (рис 4.4).

Дана функція дає доволі розгорнуту інформацію про конкретний тест, але має певний недолік, а саме поле “100% CPU” містить не правдиву інформацію, оскільки відображає загруженість тільки конкретного ядра процесори, а не процесора в цілому.

Рис. 4.4 Результат запуску окремого тесту

```
CL-USER> (parser-tests:run-test "TEST.5")
Evaluation took:
  0.023 seconds of real time
  0.023312 seconds of total run time (0.023312 user, 0.000000 system)
  100.00% CPU
  57 lambdas converted
  48,723,875 processor cycles
  2,843,792 bytes consed

T
```

Для оцінки ефективності парсингу написані три тести, які містять 10, 20 і 40 ідентичних правил граматики відповідно. Результати даного дослідження відображені в таблиці 4.1. Дослідження проводилось методом запуску кожного з тестів певної кількості раз і обчисленням середнього арифметичного значення часу виконання і кількостів циклів процесора.

Проаналізувавши результати навантажувального тестування можна зробити висновок, що залежність часу від кількості правил граматики має лінійну залежність, і завдяки цим даним можна спрогнозувати приблизну роботу парсера на довільних вхідних даних.

Таблиця 4.1 Результати навантажувального тестування

К-сть правил граматики, шт.	**Середня к-сть циклів процесора, шт.**	**Середній час виконання, с**
10	4 549 304	0.00201
20	5 990 314	0.0032
40	9 719 999	0.0041

Варто зауважити, що навантажувальне тестування проводилось на примітивних ідентичних правилах і це може вплинути на оцінку часу роботи програми на інших довільних даних.

4.2. Опис та результати інтеграційного тестування

Інтеграційне тестування проводиться для перевірки коректності взаємодії компонентів backend-частини. Такого виду тести дозволяють оцінити не тільки правильність взаємодії програмних компонентів, а й час виконання всіх трансформацій на довільних тестових даних.

Інтеграційні тести знаходяться у блоці dumper-tests і мають доволі просту і зрозумілу структуру (рисунок 4.5).

Вхідними параметрами тестового макросу є рядок вхідної граматики і список очікуваних згенерованих тестових послідовностей. Дані тести можна запускати як всі однією командою, так і окремі (рисунок 4.6) для отримання детального опису результату. Повна статистика видається у такому самому вигляді як і у блоках модульного тестування, для забезпечення однорідності тестової системи.

За допомогою статистичної інформації, яка видається блоком інтеграційного тестування можна спрогнозувати роботу програмного комплексу на довільних даних.

```
(def-test test.1
  "<A> ::= '1' | '2'
 "
  (list
   "1"
   "2"))

(def-test test.2
  "<A> ::= <B> | <C>
 <B> ::= '1'
 <C> ::= '2'
 "
  (list
   "1"
   "2"))

(def-test test.3
  "<A> ::= '1' | <B> '3'
 <B> ::= '2'
 "
  (list
   "1"
   "2 3"))
```

Рисунок 4.5 - Інтеграційні тести backend-частини

```
CL-USER> (dumper-tests::run-single-test "TEST.1")
Evaluation took:
  0.000 seconds of real time
  0.000400 seconds of total run time (0.000342 user, 0.000058 system)
  100.00% CPU
  822,738 processor cycles
  32,768 bytes consed

T
```

Рисунок 4.6 - Запуск типового інтеграційного тесту

Отже, проста структура типових інтеграційних тестів даної системи значно спрощує роботу над повним покриттям всіх блоків програмного комплексу і допомагає зробити процес розробки більш ефективним.

4.3. Опис та результати мануального тестування програмного комплексу

Мануальне тестування системи виконується безпосередньо розробником або тестувальником. Головна ідея мануального тестування – побути в ролі користувача і протестувати всі функціональні можливості конкретного застосунку. Мануальне тестування зазвичай виконується на цілісній системі.

При запуску застосунку користувач бачить перед собою вікно, яке містить чотири кнопки і два текстових поля (рисунок 4.7).

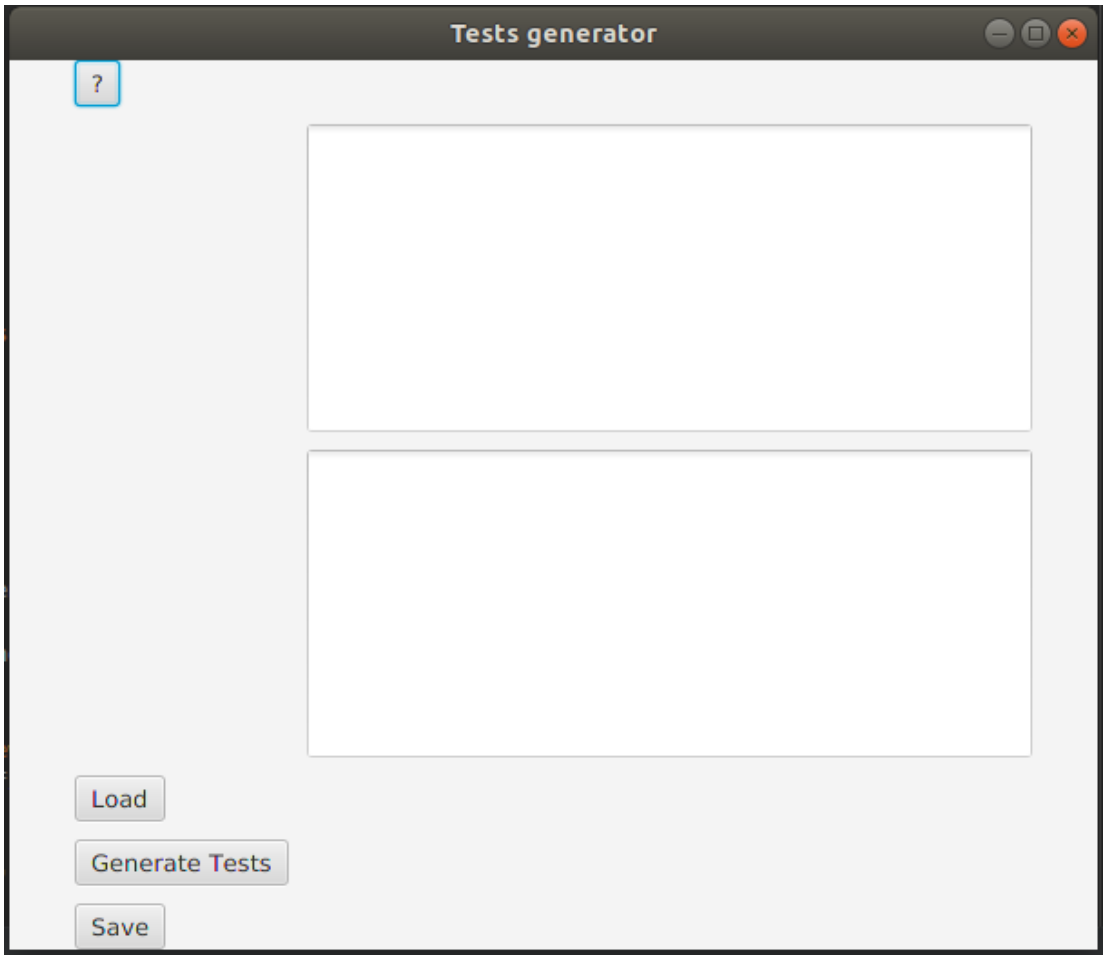
Tests generator
?
Load
Generate Tests
Save

Рисунок 4.7 - Головне вікно програми

Такий підхід до тестування важкого автоматизувати і на великих проектах такий підхід потребує значних затрат людського і часового

ресурсів. Але на невеликих проектах, або проектах, які повинні бути видані в реліз за дуже короткі терміни такий підхід повністю виправдовує себе, тому що тестовою системою являється кооперація тестувальника і продукту ПЗ.

В таблиці 4.2 описані призначення кнопок і полів.

Таблиця 4.2 Призначення елементів інтерфейсу

Назва елементу	**Призначення**
Клавіша “?”	Відкриває нове вікно, в якому користувач може отримати корисну інформацію по користуванню, опис всіх елементів або завантажити тестову граматику для ознайомлення
Клавіша “Load”	Якщо користувач не хоче власноруч писати граматику в текстовому полі номер один, то можна завантажити її з файлу, при натисненні на клавішу відкриється файловий провідник ОС користувача, після вибору вміст файлу буде відображений у текстовому полі номер один
Клавіша “Generate tests”	Виконується генерація тестів і користувач отримує результат у текстовому полі номкр два
Клавіша “Save”	Збереження результатів роботи програми у текстовому файлі, при натисненні відкривається вікно файлового провідника ОС, де можна вказати уже існуючий файл для перезапису або обрати новий

Текстове поле номер один	Призначене для зберігання вхідної граматики
Текстове поле номер два	Призначене для відображення результату роботи програми

При натисненні клавіші “?” відкриється нове вікно (рисунок 4.8). Завдяки інтуїтивно зрозумілому інтерфейсу процес ознайомлення користувача з застосунком пройде швидко і ефективно, а у разі винекнення запитань можна скористатися вікном інформаційного блоку.

Рисунок 4.8 - Інформаційне вікно користувача

Типовий мануальний тест даного програмного забезпечення виглядає наступним чином – рисунок 4.9.

Список проведених мануальних тестів:

1. Задання вхідної граматики
   a. Вручну;
   b. З файлу.
2. Виконання генерації

3. Збереження результату у файл

4. Перевірка коректності в інормаційному вікні користувача

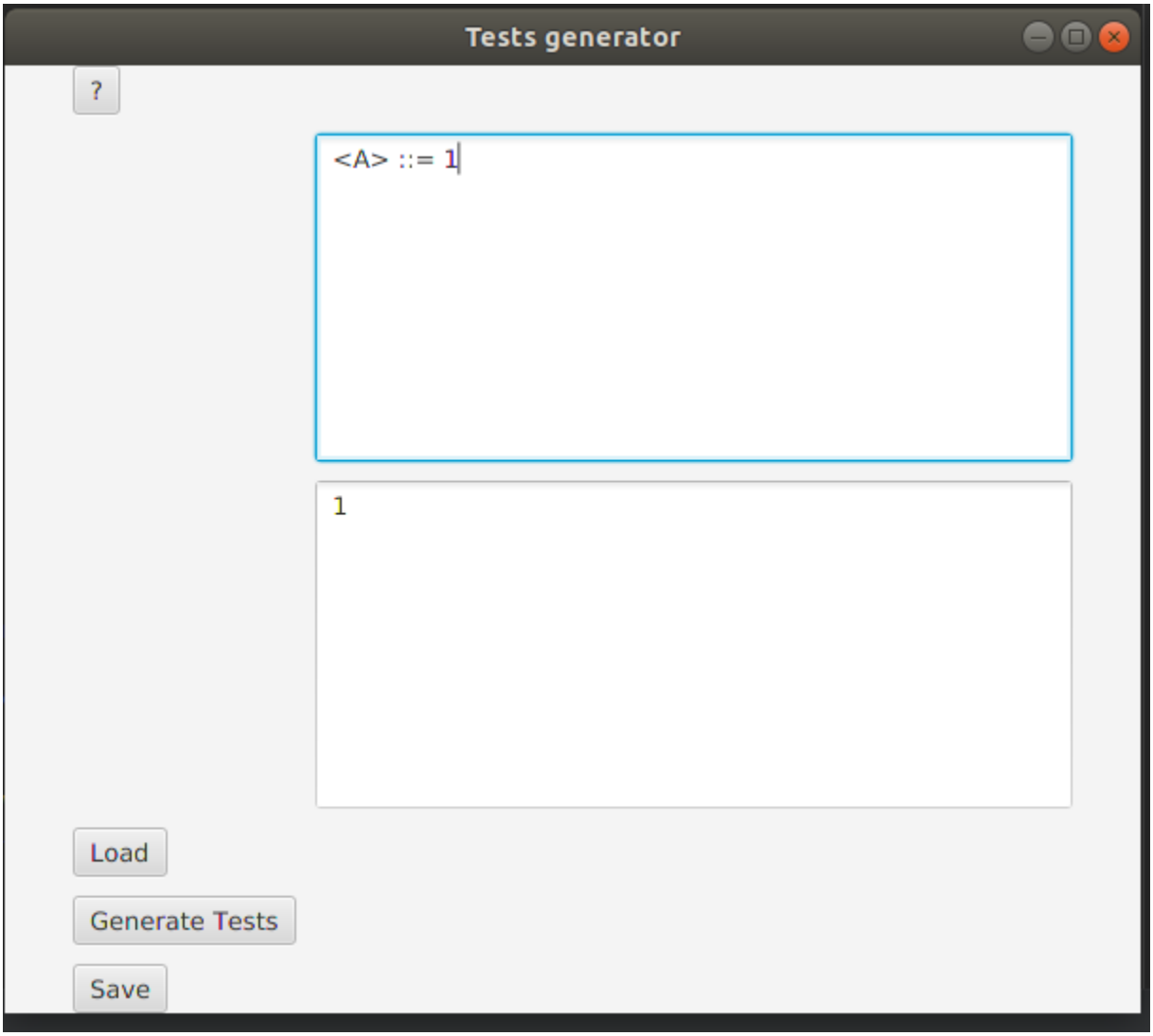

Рисунок 4.9 - Типовий мануальний тест

Отже, завдяки мануальному тестуванню можна покрити основні способи використання програмного забезпечення користувачем, але виконувати тестування логіки програмного комплексу цим методом вважається не доцільним, тому що, по-перше, дані тести доволі важко автоматизувати, по-друге, відсутність великої кількості людського ресурсу унеможливлює ефективне тестування даного проекту, тому завдяки розбиттю тестування на окремі незалежні фази, можна значно ефективніше і надійніше перевірити виконану роботу.

ВИСНОВКИ

В результаті виконаної роботи був виконаний аналіз існуючих способів і підходів автоматизації процесу тестування. На основі зібраної інформації були сформовані технічні вимоги до програмного комплексу, метою якого являється автоматизація процесу тестування трансляторів, шляхом повного покриття граматики вхідної мови.

На даний момент аналоги, які існують у відкритому доступі на ринку, не можуть справитися з даною задачею належним чином. Кожен з описаних аналогів має свої недоліки і переваги, які були враховані у розробці програмного комплексу даного курсового проєкту. Найбільший акцент ставився на швидкодії і максимально широкому спектрі використання в масштабах розглянутої сфери застосування.

Переваги даного програмного комплексу полягають в унікальності в сфері вирішення проблеми генерації тестів за заданою граматикою, висока швидкодія і широкі можливості використання. Порівнювати даний програмний комплекс у порівнянні з близькими аналогами недоцільно, тому що аналоги впринципі не можуть вирішити основної задачі, яка ставиться перед даним програмним комплексом. Але завдяки етапу навантажувального тестування можна робити прогнози роботи програмного комплексу на вхідних даних довільного розміру.

Інтуїтивно зрозумілий інтерфейс пришвидшує процес знайомства користувача з програмним продуктом, а оскільки він реалізований на мові програмування Java, то являється кросс-платформенним, що значно розширює експлуатаційні можливості. Основний функціонал реалізований на мові Lisp, що дозволяє використовувати даний програмний комплекс як в якості окремого продукту, так і в якості частини певного проекту.

Загалом дана система призначена для полегшення процесу розробки трансляторів, автоматизувавши тестову систему можна максимально ефективно розпланувати процес розробки з меншим використанням часового ресурсу.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Ахо А., Лам М. Компиляторы 2-е издание / 2008 – 1178с.

2. Caner C., Falk J. Testing computer software / Tab books 1988 -313 с.

3. Ranta A. Implementing Programming Languages / 2012 – 133 с.

4. Mogensen T. Basics of compiler design / 2010 – 319 с.

5. Марченко О. І. Основи проектування трансляторів: конспект лекцій / 2013 – 100 с.

6. Сейбел П. Практичный Common Lisp / 2005 – 482 с.